Economy of Mechanism

Sean Barnum, Cigital, Inc. [vita[3]]
Michael Gegick, Cigital, Inc. [vita[4]]

2005-09-13

One factor in evaluating a system's security is its complexity. If the design, implementation, or security mechanisms are highly complex, then the likelihood of security vulnerabilities increases. Subtle problems in complex systems may be difficult to find, especially in copious amounts of code. For instance, analyzing the source code that is responsible for the normal execution of a functionality can be a difficult task, but checking for alternate behaviors in the remaining code that can achieve the same functionality can be even more difficult. One strategy for simplifying code is the use of choke points, where shared functionality reduces the amount of source code required for an operation. Simplifying design or code is not always easy, but developers should strive for implementing simpler systems when possible.

Detailed Description Excerpts

According to Saltzer and Schroeder [Saltzer 75] in "Basic Principles of Information Protection" from page 8:

> Economy of mechanism: Keep the design as simple and small as possible. This well-known principle applies to any aspect of a system, but it deserves emphasis for protection mechanisms for this reason: design and implementation errors that result in unwanted access paths will not be noticed during normal use (since normal use usually does not include attempts to exercise improper access paths). As a result, techniques such as line-by-line inspection of software and physical examination of hardware that implements protection mechanisms are necessary. For such techniques to be successful, a small and simple design is essential.

According to Bishop [Bishop 03] in Chapter 13, "Design Principles," in "Principle of Economy of Mechanism" from pages 344-345:

> This principle simplifies the design and implementation of security mechanisms.
>
> Definition 13-3. The principle of economy of mechanism states that security mechanisms should be as simple as possible.
>
> If a design and implementation are simple, fewer possibilities exist for errors. The checking and testing process is less complex, because fewer components and cases need to be tested. Complex mechanisms often make assumptions about the system and environment in which they run. If these assumptions are incorrect, security problems may result.
>
> Interfaces to other modules are particularly suspect, because modules often make implicit assumptions about input or output parameters or the current system state; should any of these assumptions be wrong, the module's actions may produce unexpected, and erroneous, results. Interaction with external entities, such as other programs, systems, or humans, amplified this

[3] daisy:35 (Barnum, Sean)

[4] daisy:345 (Gegick, Michael)

problem.

> **Example 1**
>
> The ident protocol sends the user name associated with a process that has a TCP connection to a remote host. A mechanism on host A that allows access based on the results of an ident protocol result makes the assumption that the originating host is trustworthy. If host B decides to attack host A, it can connect and then send any identity it chooses in response to the ident request. This is an example of a mechanism making an incorrect assumption about the environment (specifically that host B can be trusted).
>
> **Example 2**
>
> The finger protocol transmits information about a user or system. Many client implementations assume that the server's response is well-formed. However, if an attacker were to create a server that generated an infinite stream of characters, and a finger client were to connect to it, the client would print all the characters. As a result, log files and disks could be filled up, resulting in a denial of service attack on the querying host. this is an example of incorrect assumptions about the input to the client.

According to Viega and McGraw [Viega 02] in Chapter 5, "Guiding Principles for Software Security," in "Principle 6: Keep It Simple" from pages 104-107:

> The KISS mantra is pervasive: "Keep It Simple, Stupid!" This motto applies just as well to security as it does everywhere else. Complexity increases the risk of problems. Avoid complexity and avoid problems.
>
> The most obvious implication is that software design and implementation should be as straightforward as possible. Complex design is never easy to understand, and is therefore more likely to include subtle problems that will be missed during analysis. Complex code tends to be harder to maintain as well. And most importantly, complex software tends to be far more buggy. We don't think this comes as any big surprise to anyone.
>
> Consider reusing components whenever possible, as long as the components to be reused are believed to be of good quality. The more successful use that a particular component has seen, the more intent you should be on not having to rewrite it. This consideration particularly holds true for cryptographic libraries. Why would anyone want to re-implement AES or SHA-1, when there are several widely used libraries available? Well-used libraries are much more likely to be robust than something put together in-house, since people are more likely to have noticed implementation problems. Experience builds assurance, especially when those experiences are positive. Of course, there is always the possibility of problems even in widely used components, but it's reasonable to suspect that there's less risk involved in the known quantity, all other things [being] equal (although, do refer back to Chapter 4 [in *Building Secure Software*] for several caveats).
>
> It also stands to reason that adding bells and whistles tends to violate the simplicity principle. True enough. But what if the bells and whistles in question are security features? When we discussed defense in depth, we said that we wanted redundancy. Here, we seem to be arguing the opposite. We previously said, "don't put all your eggs in one basket." Now we're saying, "be wary of having multiple baskets." Both notions make sense, even though they're obviously at odds with each other.
>
> The key to unraveling this paradox is to strike a balance that is right for each particular project. When adding redundant features, the idea is to improve the apparent security of the system. Once

enough redundancy has been added to address whatever security level is desired, then extra redundancy is not necessary. In practice, a second layer of defense is usually a good idea, but a third layer should be carefully considered.

Despite its face value obviousness, the simplicity principle has its subtleties. Building as simple a system as possible while still meeting security requirements is not always easy. An online trading system without encryption is certainly simpler than an otherwise equivalent one that includes crypto, but there's no way that it's more secure.

Simplicity can often be improved by funneling all security-critical operations through a small number of choke points in a system. The idea behind a choke point is to create a small, easily controlled interface through which control must pass. This is one way to avoid spreading security code throughout a system. In addition, it is far easier to monitor user behavior and input if all users are forced into a few small channels. That's the idea behind having only a few entrances at sports stadiums; if there were too many entrances, collecting tickets would be harder and more staff would be required to do the same quality job.

One important thing about choke points is that there should be no secret ways around them. Harking back to our example, if a stadium has an unsecured chain link fence, you can be sure that people without tickets will climb it. Providing "hidden" administrative functionality or "raw" interfaces to your functionality that are available to savvy attackers can easily backfire. There have been plenty of examples where a hidden administrative backdoor could be used by a knowledgeable intruder, such as a backdoor in the Dansie shopping cart, or a backdoor in Microsoft FrontPage, both discovered in the same month (April, 2000). The FrontPage backdoor became somewhat famous due to a hidden encryption key that read, "Netscape engineers are weenies!"

Another not-so-obvious but important aspect of simplicity is usability. Anyone who needs to use a system should be able to get the best security it has to offer easily, and should not be able to introduce insecurities without thinking carefully about it. Even then, they should have to bend over backwards. Usability applies both to the people who use a program, and to the people who have to maintain its code base, or program against its API.

Many people seem to think they've got an intuitive grasp on what is easy to use. But usability tests tend to prove them wrong. That might be okay for generic functionality, because a given product might be cool enough that ease of use isn't a real concern. When it comes to security, though, usability becomes more important than ever.

Strangely enough, there's an entire science of usability. All software designers should read two books in this field, *The Design of Everyday Things*[8] and *Usability Engineering.*[9] This space is too small to give adequate coverage to the topic. However, we can give you some tips, as they apply directly to security:

1. The user will not read documentation. If the user needs to do anything special to get the full benefits of the security you provide, then the user is unlikely to receive those benefits. Therefore, you should provide security by default. A user should not need to know anything about security or have to do anything in particular to be able to use your solution securely. Of course, as security is a relative term, you will have to make some decisions as to the security requirements of your users.

   Consider enterprise application servers that have encryption turned off by default. Such functionality is usually turned on with a menu option on an administrative tool somewhere. But even if a system administrator stops to think about security, it's likely that person will

[8] Norman, Donald A. *The Design of Everyday Things*. New York, NY: Basic Books, 2002.

[9] Nielsen, Jakob. *Usability Engineering*. San Francisco, CA: Morgan Kaufmann, 1994.

think, "they certainly have encryption on by default."

2. Talk to users to figure out what their security requirements are. As Jakob Nielson likes to say, corporate vice presidents are NOT users. You shouldn't assume that you know what people will need; go directly to the source. Try to provide users with more security than they think they need.

3. Realize that users aren't always right. Most users aren't well informed about security. They may not understand many security issues, so try to anticipate their needs. Don't give them security dialogs that they can ignore. Err on the side of security. If your assessment of their needs provides more security than theirs, use yours (actually, try to provide more security than you think they need in either case).

   As an example, think about a system such as a web portal, where one service you provide is stock quotes. Your users might never think there's a good reason to secure that kind of stuff at all. After all, stock quotes are for public consumption. But there is a good reason—an attacker could tamper with the quotes users get. Users might decide to buy or sell something based on bogus information, and lose their shirts. Sure, you don't have to encrypt the data; you can use a MAC (Message Authentication Code; see Appendix A [in *Building Secure Software*]). But most users are unlikely to anticipate this risk.

4. Users are lazy. They're so lazy that they won't actually stop to consider security, even when you throw up a dialog box that says "WARNING!" in big, bright red letters. To the user, dialog boxes are an annoyance if they keep the user from what he or she wants to do. For example, a lot of mobile code systems (such as web based ActiveX controls) will pop up a dialog box, telling you who signed the code, and asking you if you really want to trust that signer. Do you think anyone actually reads that stuff? Nope. Users just want to get to see the program advertised run, and will take the path of least resistance, without considering the consequences. In the real world, the dialog box is just a big annoyance. As Ed Felten says, "Given the choice between dancing pigs and security, users will pick dancing pigs every time."

> Keeping it simple is important in many domains, including security.

We include three relevant principles from NIST [NIST 01] in Section 3.3, "IT Security Principles," from page 16:

> **Do not implement unnecessary security mechanisms.** Every security mechanism should support a security service or set of services, and every security service should support one or more security goals. Extra measures should not be implemented if they do not support a recognized service or security goal. Such mechanisms could add unneeded complexity to the system and are potential sources of additional vulnerabilities. An example is file encryption supporting the access control service that in turn supports the goals of confidentiality and integrity by preventing unauthorized file access. If file encryption is a necessary part of accomplishing the goals, then the mechanism is appropriate. However, if these security goals are adequately supported without inclusion of file encryption, then that mechanism would be an unneeded system complexity.

From page 10:

> **Strive for Simplicity.** The more complex the mechanism, the more likely it may possess exploitable flaws. Simple mechanisms tend to have fewer exploitable flaws and require less maintenance. Further, because configuration management issues are simplified, updating or replacing a simple mechanism becomes a less intensive process.

From page 17:

> **Strive for operational ease of use.** The more difficult it is to maintain and operate a security control, the less effective that control is likely to be. Therefore, security controls should be designed with ease-of-use as an important consideration. The experience and expertise of administrators and users should be appropriate and proportional to the operation of the security control. An organization must invest the resources necessary to ensure system administrators and users are properly trained. Moreover, administrator and user training costs along with the life-cycle operational costs should be considered when determining the cost-effectiveness of the security control.

According to Schneier [Schneier 00] in the section "Security Processes":

> **Embrace Simplicity.** Keep things as simple as absolutely possible. Security is a chain; the weakest link breaks it. Simplicity means fewer links.

According to McGraw and Viega [McGraw 03]:[10]

> **Security risks can remain hidden due to a system's complexity.** By their very nature, complex systems introduce multiple risks—and almost all systems that involve software are complex. One risk is that malicious functionality can be added to a system (either during creation or afterward) that extends it past its intended design. As an unfortunate side effect, inherent complexity lets malicious and flawed subsystems remain invisible to unsuspecting users until it's too late. This is one of the root causes of the malicious code problem. Another risk, more relevant to our purposes, is that a system's complexity makes it hard to understand, hard to analyze and hard to secure. Security is difficult to get right even in simple systems; complex systems serve only to make security more difficult. Security risks can remain hidden in the jungle of complexity, not coming to light until these areas have been exploited.
>
> A desktop system running Windows/XP and associated applications depends upon the proper functioning of the kernel as well as the applications to ensure that vulnerabilities can't compromise the system. However, XP itself consists of at least 40 million lines of code, and end-user applications are becoming equally, if not more, complex. When systems become this large, bugs can't be avoided.
>
> The complexity problem is exacerbated by the use of unsafe programming languages (for example, C or C++) that don't protect against simple kinds of attacks, such as buffer overflows. In theory, we can analyze and prove that a small program is free of problems, but this task is impossible for even the simplest desktop systems today, much less the enterprise-wide systems employed by businesses or governments.

References

[Bishop 03]	Bishop, Matt. *Computer Security: Art and Science*. Boston, MA: Addison-Wesley, 2003.
[McGraw 03]	McGraw, Gary & Viega, John. "Keep It Simple." *Software Development.* CMP Media LLC, May, 2003.
[NIST 01]	*Engineering Principles for Information Technology Security* . Special Publication 800-27. US Department of Commerce, National Institute of Standards and Technology, 2001.

[10] All rights reserved. It is reprinted with permission from CMP Media LLC.

[Saltzer 75]	Saltzer, Jerome H. & Schroeder, Michael D. "The Protection of Information in Computer Systems," 1278-1308. *Proceedings of the IEEE 63*, 9. IEEE, September 1975.
[Schneier 00]	Schneier, Bruce. "The Process of Security." *Information Security Magazine.* April, 2000. http://infosecuritymag.techtarget.com/archives2000.shtml#apr2000.
[Viega 02]	Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way*. Boston, MA: Addison-Wesley, 2002.

Cigital, Inc. Copyright

####

###	########
Copyright Holder	Cigital, Inc.

####

###	########
is-content-area-overview	false
Content Areas	Knowledge/Principles
SDLC Relevance	Design
Workflow State	Publishable

[1] mailto:copyright@cigital.com